Programming Languages

2nd edition

Tucker and Noonan

Chapter 2
Syntax

A language that is simple to parse for the compiler is also simple to parse for the human programmer.

N. Wirth

Contents

Thinking about Syntax

- The *syntax* of a programming language is a precise description of all its grammatically correct programs.
- Precise syntax was first used with Algol 60, and has been used ever since.
- Three levels:
  - *Lexical syntax*
  - *Concrete syntax*
  - *Abstract syntax*

Levels of Syntax

- Lexical syntax = all the basic symbols of the language (names, values, operators, etc.)
- Concrete syntax = rules for writing expressions, statements and programs.
- Abstract syntax = internal representation of the program, favoring content over form. E.g.,
  - *C:* *if (expr) ...* *discard ( )*
  - *Ada:* *if (expr) then* *discard then*

2.1 Grammars

- A *metalanguage* is a language used to define other languages.
- A *grammar* is a metalanguage used to define the syntax of a language.
- *Our interest*: using grammars to define the syntax of a programming language.

2.1.1 Backus-Naur Form (BNF)

- Stylized version of a context-free grammar (cf. Chomsky hierarchy)
- Sometimes called Backus Normal Form
- First used to define syntax of Algol 60
- Now used to define syntax of most major languages

BNF Grammar

- Set of *productions*: P
- *terminal* symbols: T
- *nonterminal* symbols: N
- *start* symbol: S

- A *production* has the form
- $A \rightarrow B$
- where and

Example: Binary Digits

- Consider the grammar:

$$binaryDigit \rightarrow 0$$
$$binaryDigit \rightarrow 1$$

- or equivalently:

$$binaryDigit \rightarrow 0 \mid 1$$

- Here, | is a metacharacter that separates alternatives.

2.1.2 Derivations

- Consider the grammar:

  $Integer \rightarrow Digit \mid Integer\ Digit$

  $Digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

- We can *derive* any unsigned integer, like 352, from this grammar.

Derivation of 352 as an *Integer*

- A 6-step process, starting with:

Integer

Derivation of 352 (step 1)

- Use a grammar rule to enable each step:

$$Integer \Rightarrow Integer\ Digit$$

Derivation of 352 (steps 1-2)

- Replace a nonterminal by a right-hand side of one of its rules:

$$\begin{aligned} Integer &\Rightarrow Integer\ Digit \\ &\Rightarrow Integer\ 2 \end{aligned}$$

Derivation of 352 (steps 1-3)

- Each step follows from the one before it.

$$\begin{aligned} \mathit{Integer} &\Rightarrow \mathit{Integer}\ \mathit{Digit} \\ &\Rightarrow \mathit{Integer}\ 2 \\ &\Rightarrow \mathit{Integer}\ \mathit{Digit}\ 2 \end{aligned}$$

Derivation of 352 (steps 1-4)

$$\begin{aligned} Integer &\Rightarrow Integer\ Digit \\ &\Rightarrow Integer\ 2 \\ &\Rightarrow Integer\ Digit\ 2 \\ &\Rightarrow Integer\ 5\ 2 \end{aligned}$$

Derivation of 352 (steps 1-5)

$$\begin{aligned} Integer &\Rightarrow Integer\ Digit \\ &\Rightarrow Integer\ 2 \\ &\Rightarrow Integer\ Digit\ 2 \\ &\Rightarrow Integer\ 5\ 2 \\ &\Rightarrow Digit\ 5\ 2 \end{aligned}$$

Derivation of 352 (steps 1-6)

- You know you're finished when there are only terminal symbols remaining.

$$
\begin{aligned}
\mathit{Integer} &\Rightarrow \mathit{Integer}\ \mathit{Digit} \\
&\Rightarrow \mathit{Integer}\ 2 \\
&\Rightarrow \mathit{Integer}\ \mathit{Digit}\ 2 \\
&\Rightarrow \mathit{Integer}\ 5\ 2 \\
&\Rightarrow \mathit{Digit}\ 5\ 2 \\
&\Rightarrow 3\ 5\ 2
\end{aligned}
$$

A Different Derivation of 352

$$\begin{aligned} \mathit{Integer} &\Rightarrow \mathit{Integer\ Digit} \\ &\Rightarrow \mathit{Integer\ Digit\ Digit} \\ &\Rightarrow \mathit{Digit\ Digit\ Digit} \\ &\Rightarrow 3\ \mathit{Digit\ Digit} \\ &\Rightarrow 3\ 5\ \mathit{Digit} \\ &\Rightarrow 3\ 5\ 2 \end{aligned}$$

- This is called a *leftmost derivation*, since at each step the leftmost nonterminal is replaced.
- (The first one was a *rightmost derivation*.)

Notation for Derivations

- $Integer \Rightarrow^* 352$

  Means that 352 can be derived in a finite number of steps using the grammar for *Integer*.

- $352 \in L(G)$

  Means that 352 is a member of the language defined by grammar *G*.

- $L(G) = \{\ \omega \in T^* \mid Integer \Rightarrow^* \omega\ \}$

  Means that the language defined by grammar *G* is the set of all symbol strings ω that can be derived as an *Integer*.

2.1.3 Parse Trees

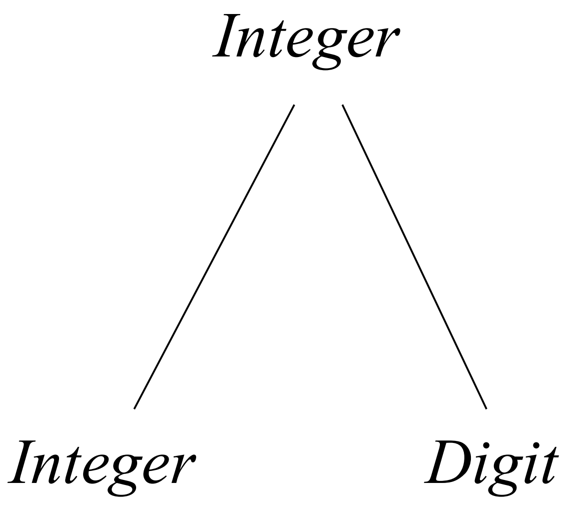
- A *parse tree* is a graphical representation of a derivation.

  Each internal node of the tree corresponds to a step in the derivation.

  Each child of a node represents a right-hand side of a production.

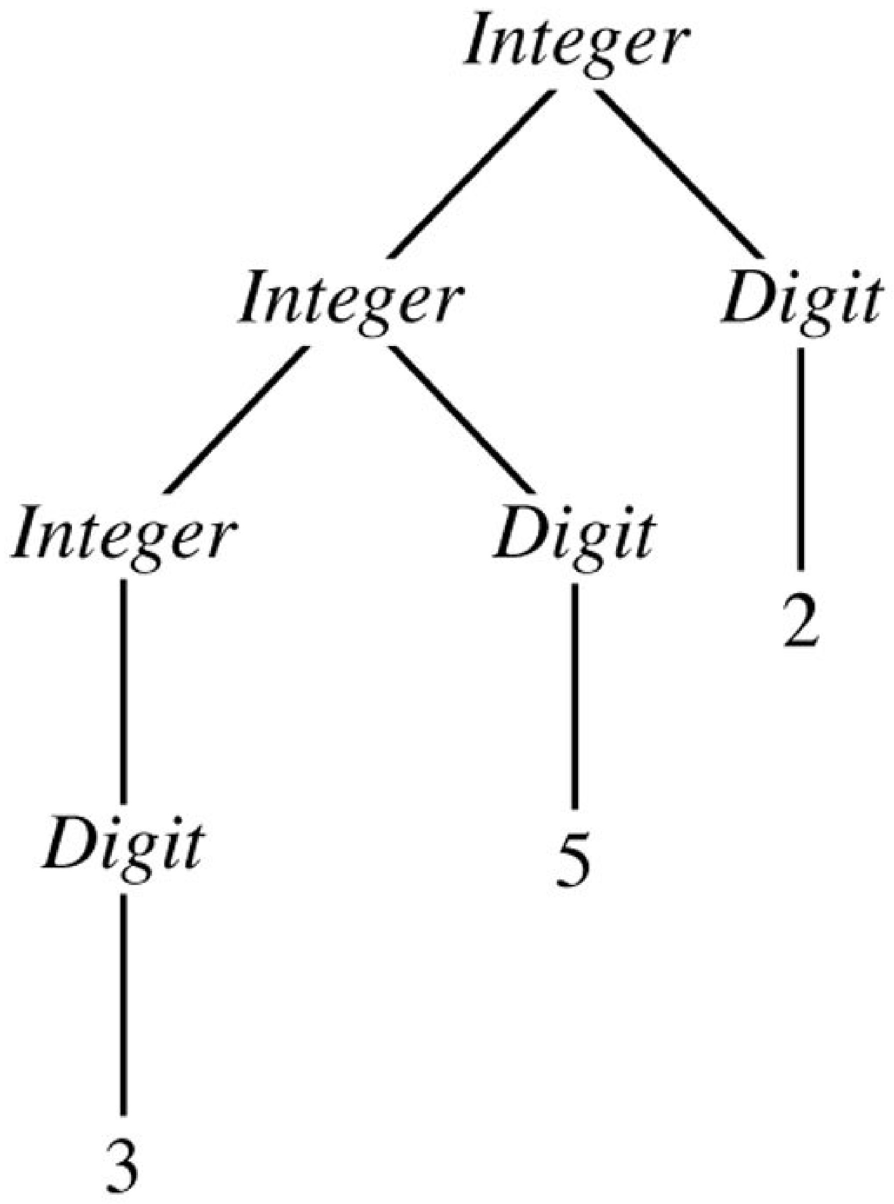
  Each leaf node represents a symbol of the derived string, reading from left to right.

E.g., The step *Integer* $\Rightarrow$ *Integer Digit* appears in the parse tree as:



Parse Tree for 352 as an *Integer*

Figure 2.1



Arithmetic Expression Grammar

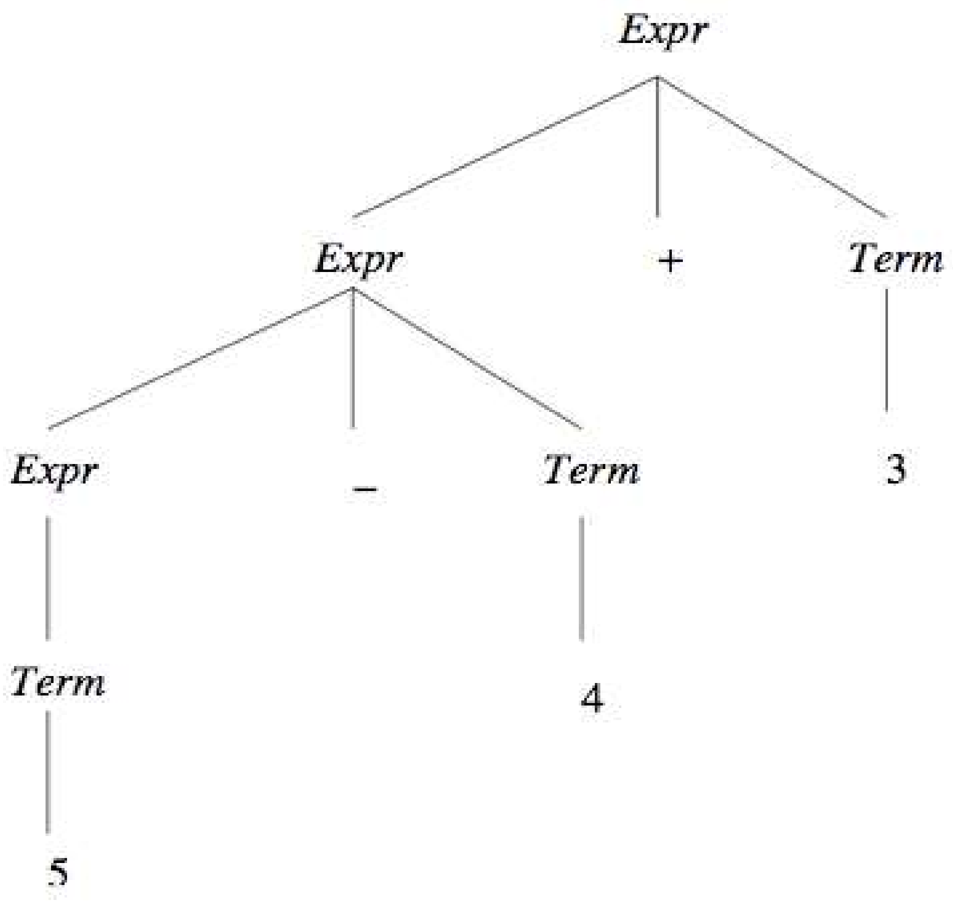
- The following grammar defines the language of arithmetic expressions with 1-digit integers, addition, and subtraction.

$$Expr \rightarrow Expr + Term \mid Expr - Term \mid Term$$

$$Term \rightarrow 0 \mid \ldots \mid 9 \mid (\; Expr \;)$$

Parse of the String 5-4+3

Figure 2.2



2.1.4 Associativity and Precedence

- A grammar can be used to define associativity and precedence among the operators in an expression.

  E.g., + and - are left-associative operators in mathematics;

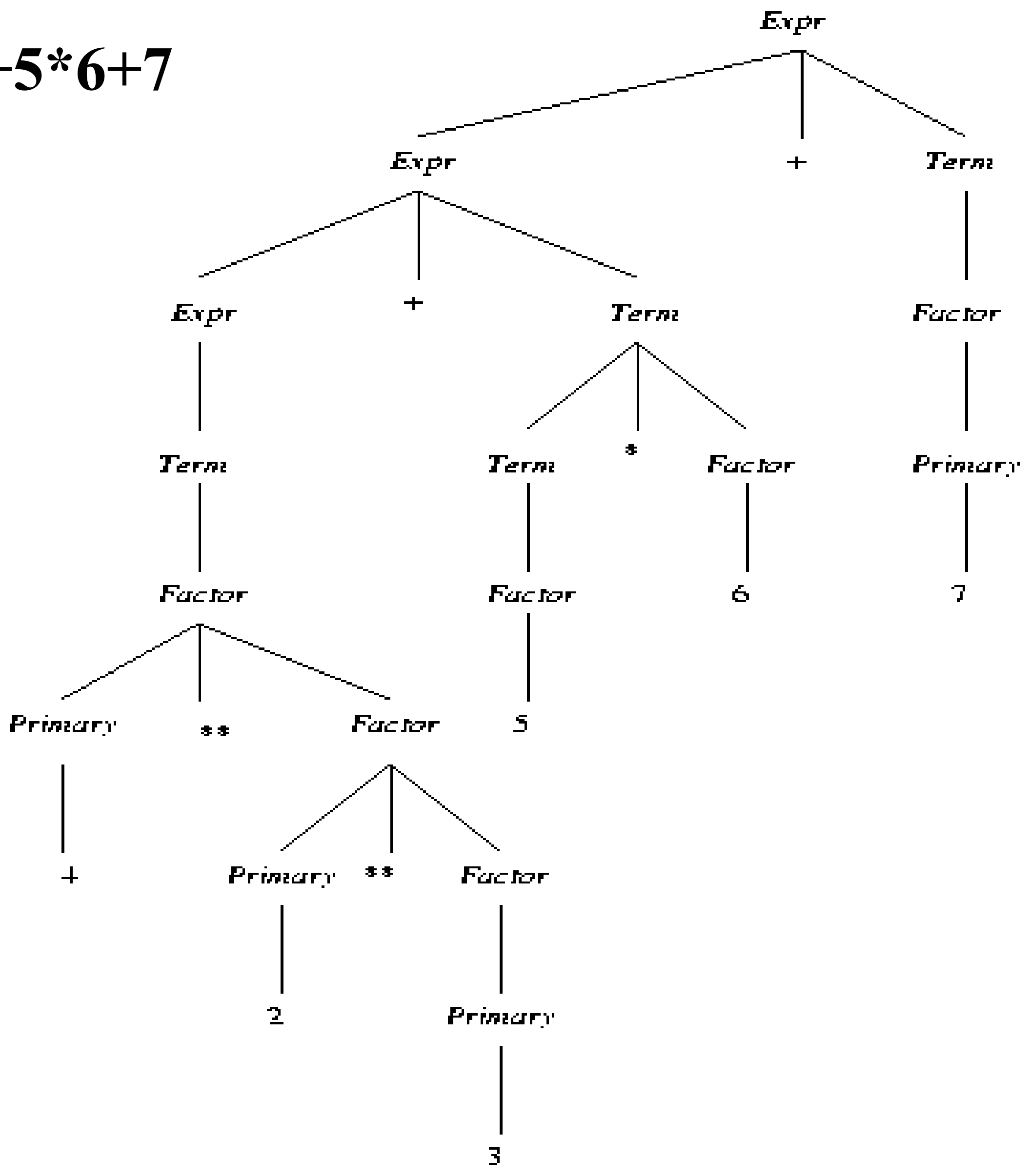
  ** and / have higher precedence than + and - .*

- Consider the more interesting grammar G_1:

  Expr -> Expr + Term | Expr – Term | Term
  *Term -> Term * Factor | Term / Factor | Term % Factor | Factor*
  *Factor -> Primary ** Factor | Primary*
  Primary -> 0 | ... | 9 | (Expr)

Parse of 4**2**3+5*6+7 for Grammar G_1

Figure 2.3



Associativity and Precedence for Grammar G_1

Table 2.1

Precedence	Associativity	Operators
3	right	**
2	left	* / %
1	left	+ -

•*Note: These relationships are shown by the structure of the parse tree: highest precedence at the bottom, and left-associativity on the left at each level.*

2.1.5 Ambiguous Grammars

- A grammar is *ambiguous* if one of its strings has two or more diffferent parse trees.

  E.g., Grammar G$_1$ *above is unambiguous.*

- C, C++, and Java have a large number of
  - *operators and*
  - *precedence levels*

- Instead of using a large grammar, we can:
  - *Write a smaller ambiguous grammar, and*
  - *Give separate precedence and associativity (e.g., Table 2.1)*

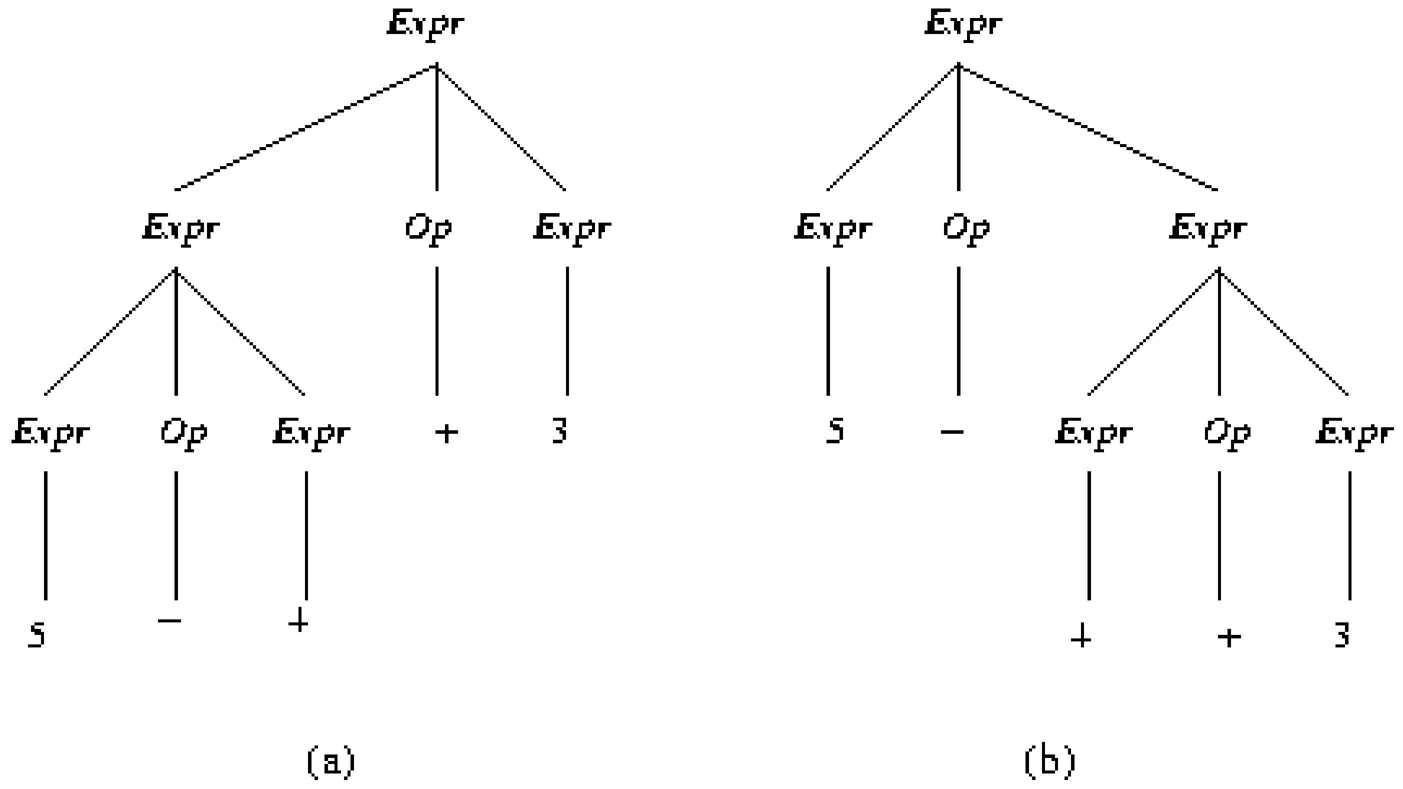
An Ambiguous Expression Grammar G_2

- *Expr -> Expr Op Expr* | (*Expr*) | *Integer*
- *Op ->* + | - | * | / | % | **

- Notes:
  - G_2 is equivalent to G_1. I.e., its language is the same.
  - G_2 has fewer productions and nonterminals than G_1.
  - However, G_2 is ambiguous.

Ambiguous Parse of 5-4+3 Using Grammar G_2

Figure 2.4



The Dangling Else

IfStatement -> if (*Expression*) *Statement* |

if (*Expression*) *Statement* else *Statement*

Statement -> *Assignment* | *IfStatement* | *Block*

Block -> { *Statements* }

Statements -> *Statements* *Statement* | *Statement*

Example

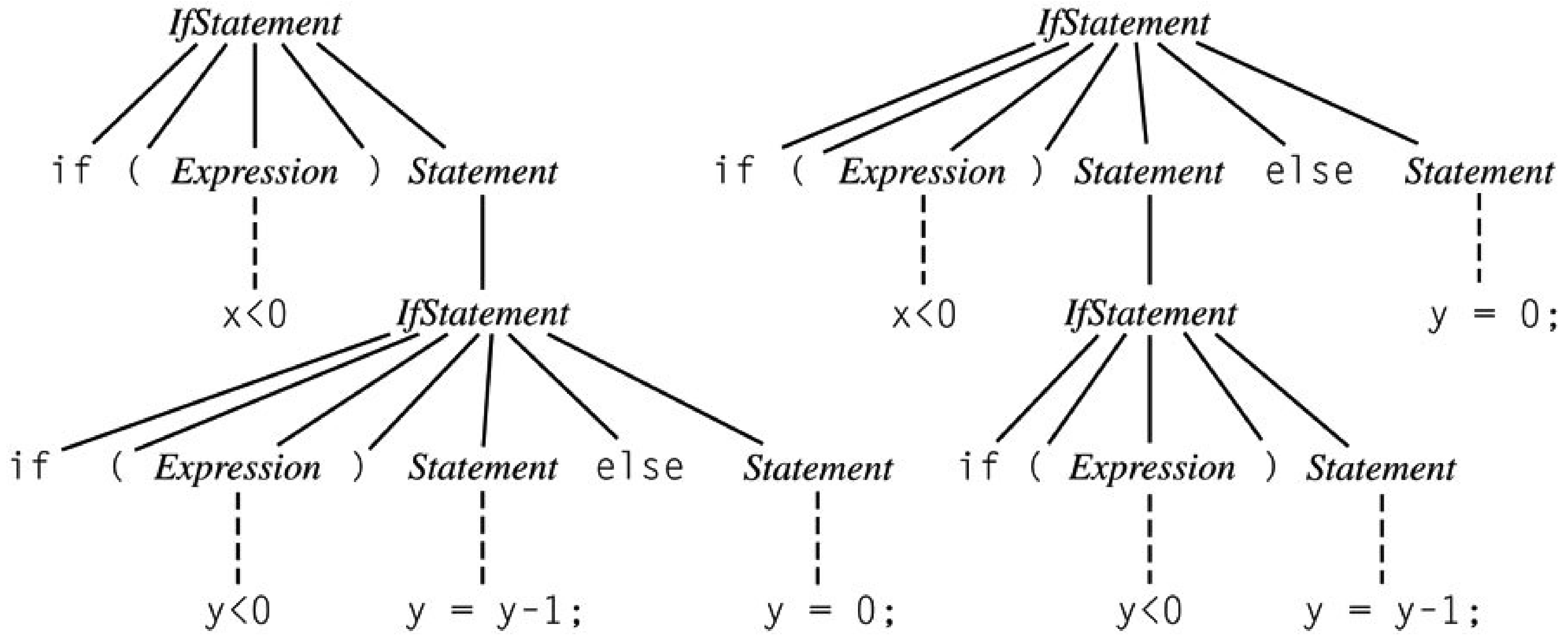
•With which ‘if’ does the following ‘else’ associate

```
if (x < 0)
    if (y < 0)  y = y  - 1;
    else y = 0;
```

•Answer: *either one!*

The *Dangling Else* Ambiguity

Figure 2.5


IfStatement
if (Expression) Statement
x<0
IfStatement
if (Expression) Statement else Statement
y<0
y = y-1;
y = 0;
IfStatement
if (Expression) Statement else Statement
x<0
IfStatement
y = 0;
if (Expression) Statement
y<0
y = y-1;

Solving the dangling else ambiguity

1. Algol 60, C, C++: associate each else with closest if; use {} or begin…end to override.
2. Algol 68, Modula, Ada: use explicit delimiter to end every conditional (e.g., if…fi)
3. Java: rewrite the grammar to limit what can appear in a conditional:

```
IfThenStatement -> if ( Expression ) Statement
IfThenElseStatement -> if ( Expression ) StatementNoShortIf
                                     else Statement
```

The category *StatementNoShortIf* includes all except *IfThenStatement*.

2.2 Extended BNF (EBNF)

- BNF:
  - *recursion for iteration*
  - *nonterminals for grouping*
- EBNF: additional metacharacters
  - **{ }** for a series of zero or more
  - **( )** for a list, must pick one
  - **[ ]** for an optional list; pick none or one

EBNF Examples

- *Expression* is a list of one or more *Terms* separated by operators + and -

  Expression -> *Term* **{ (+ | -)** *Term* **}**

  IfStatement -> if (*Expression*) *Statement* **[** else *Statement* **]**

- *C-style EBNF lists alternatives vertically and uses* $_{opt}$ *to signify optional parts. E.g.,*

  IfStatement:

  if (*Expression*) *Statement* *ElsePart*$_{opt}$

  ElsePart:

  else *Statement*

EBNF to BNF

•We can always rewrite an EBNF grammar as a BNF grammar. E.g.,

A -> *x* { *y* } *z*

can be rewritten:

A -> *x A' z*

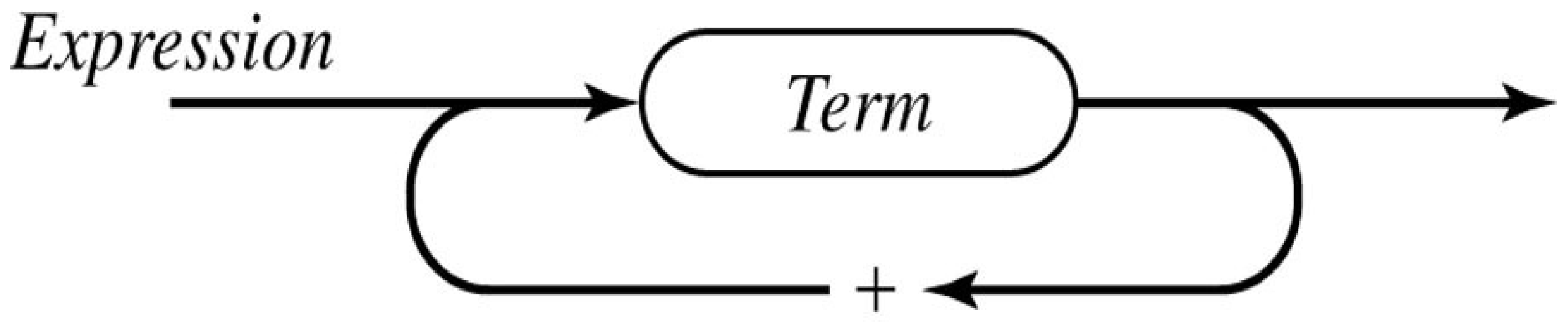
A' -> | *y A'*

(Rewriting EBNF rules with ( ), [ ] is left as an exercise.)

•*While EBNF is no more powerful than BNF, its rules are often simpler and clearer.*

Syntax Diagram for *Expressions* with Addition

Figure 2.6



2.3 Syntax of a Small Language: *Clite*

- Motivation for using a subset of C:

Language	*Grammar (pages)*	*Reference*
Pascal	5	Jensen & Wirth
C	6	Kernighan & Richie
C++	22	Stroustrup
Java	14	Gosling, et. al.

- The *Clite* grammar fits on one page (next 3 slides),
- so it’s a far better tool for studying language design.

Fig. 2.7 *Clite* Grammar: Statements

$$\begin{aligned}
Program &\rightarrow \texttt{int main ( ) \{ } Declarations\ Statements\ \texttt{\}} \\
Declarations &\rightarrow \{\ Declaration\ \} \\
Declaration &\rightarrow Type\ Identifier\ [\ \texttt{[}\ Integer\ \texttt{]}\ ]\ \{\ \texttt{,}\ Identifier\ [\ \texttt{[}\ Integer\ \texttt{]}\ ]\ \} \\
Type &\rightarrow \texttt{int} \mid \texttt{bool} \mid \texttt{float} \mid \texttt{char} \\
Statements &\rightarrow \{\ Statement\ \} \\
Statement &\rightarrow \texttt{;} \mid Block \mid Assignment \mid IfStatement \mid WhileStatement \\
Block &\rightarrow \texttt{\{}\ Statements\ \texttt{\}} \\
Assignment &\rightarrow Identifier\ [\ \texttt{[}\ Expression\ \texttt{]}\ ]\ \texttt{=}\ Expression\ \texttt{;} \\
IfStatement &\rightarrow \texttt{if (}\ Expression\ \texttt{)}\ Statement\ [\ \texttt{else}\ Statement\ ] \\
WhileStatement &\rightarrow \texttt{while (}\ Expression\ \texttt{)}\ Statement
\end{aligned}$$

Fig. 2.7 *Clite* Grammar: Expressions

$$\mathit{Expression} \rightarrow \mathit{Conjunction}\ \{\ \texttt{||}\ \ \mathit{Conjunction}\ \}$$

$$\mathit{Conjunction} \rightarrow \mathit{Equality}\ \{\ \texttt{\&\&}\ \ \mathit{Equality}\ \}$$

$$\mathit{Equality} \rightarrow \mathit{Relation}\ [\ \mathit{EquOp}\ \mathit{Relation}\ ]$$

$$\mathit{EquOp} \rightarrow \ \texttt{==}\ \mid\ \texttt{!=}$$

$$\mathit{Relation} \rightarrow \mathit{Addition}\ [\ \mathit{RelOp}\ \mathit{Addition}\ ]$$

$$\mathit{RelOp} \rightarrow \ \texttt{<}\ \mid\ \texttt{<=}\ \mid\ \texttt{>}\ \mid\ \texttt{>=}$$

$$\mathit{Addition} \rightarrow \mathit{Term}\ \{\ \mathit{AddOp}\ \mathit{Term}\ \}$$

$$\mathit{AddOp} \rightarrow \ \texttt{+}\ \mid\ \texttt{-}$$

$$\mathit{Term} \rightarrow \mathit{Factor}\ \{\ \mathit{MulOp}\ \mathit{Factor}\ \}$$

$$\mathit{MulOp} \rightarrow \ \texttt{*}\ \mid\ \texttt{/}\ \mid\ \texttt{\%}$$

$$\mathit{Factor} \rightarrow [\ \mathit{UnaryOp}\ ]\ \mathit{Primary}$$

$$\mathit{UnaryOp} \rightarrow \ \texttt{-}\ \mid\ \texttt{!}$$

$$\mathit{Primary} \rightarrow \mathit{Identifier}\ [\ \ [\ \mathit{Expression}\ ]\ ]\ \mid \mathit{Literal} \mid\ (\ \mathit{Expression}\ )\ \mid \mathit{Type}\ (\ \mathit{Expression}\ )$$

Fig. 2.7 *Clite* grammar: lexical level

$Identifier \rightarrow Letter \; \{ \; Letter \mid Digit \; \}$

$Letter \rightarrow$ `a | b | … | z | A | B | … | Z`

$Digit \rightarrow$ `0 | 1 | … | 9`

$Literal \rightarrow Integer \mid Boolean \mid Float \mid Char$

$Integer \rightarrow Digit \; \{ \; Digit \; \}$

$Boolean \rightarrow$ `true | False`

$Float \rightarrow Integer$ `.` $Integer$

$Char \rightarrow$ `'` $ASCII \; Char$ `'`

Issues Not Addressed by this Grammar

- Comments
- Whitespace
- Distinguishing one token **<=** from two tokens **< =**
- Distinguishing identifiers from keywords like **if**

- These issues are addressed by identifying two levels:
  - *lexical level*
  - *syntactic level*

2.3.1 Lexical Syntax

- *Input*: a stream of characters from the ASCII set, keyed by a programmer.
- *Output*: a stream of *tokens* or basic symbols, classified as follows:

– *Identifiers*	e.g., Stack, x, i, push
– *Literals*	e.g., 123, 'x', 3.25, true
– *Keywords*	bool char else false float if int main true while
– *Operators*	= || && == != < <= > >= + - * / !
– *Punctuation*	; , { } ( )

Whitespace

- Whitespace is any space, tab, end-of-line character (or characters), or character sequence inside a comment
- No token may contain embedded whitespace
  - *(unless it is a character or string literal)*
- Example:

  >= *one token*

  > = *two tokens*

Whitespace Examples in Pascal

- `while  a  <  b  do` *legal* - spacing between tokens
- `while  a<b  do` spacing not needed for <

- `whilea<bdo` *illegal* - can’t tell boundaries
- `whilea  <  bdo` between tokens

Comments

- Not defined in grammar
- *Clite* uses // comment style of C++

Identifier

- Sequence of letters and digits, starting with a letter
  - *`if` is both an identifier and a keyword*
  - *Most languages require identifiers to be distinct from keywords*

- In some languages, identifiers are merely predefined (and thus can be redefined by the programmer)

Redefining Identifiers can be dangerous

```
program confusing;
const true = false;
begin
    if  (a<b) = true then f(a)
    else …
```

Should Identifiers be case-sensitive?

- Older languages: no. Why?
  - *Pascal: no.*
  - *Modula: yes*
  - *C, C++: yes*
  - *Java: yes*
  - *PHP: partly yes, partly no. What about orthogonality?*

2.3.2 Concrete Syntax

- Based on a parse of its *Tokens*
  - *; is a statement terminator*
  - *(Algol-60, Pascal use ; as a separator)*

- Rule for *IfStatement* is ambiguous:

  "The else ambiguity is resolved by connecting an **else** with the last encountered else-less if."

  [Stroustrup, 1991]

Expressions in *Clite*

- 13 grammar rules
- Use of meta braces – operators are left associative
- C++ expressions require 4 pages of grammar rules [Stroustrup]
- C uses an ambiguous expression grammar [Kernighan and Ritchie]

Associativity and Precedence

Clite Operator	Associativity
Unary - !	none
* /	left
+ -	left
< <= > >=	none
== !=	none
&&	left
||	left

Clite Equality, Relational Operators

- … are non-associative.
  (an idea borrowed from Ada)

- Why is this important?

In C++, the expression:

```
  if (a < x < b)
```

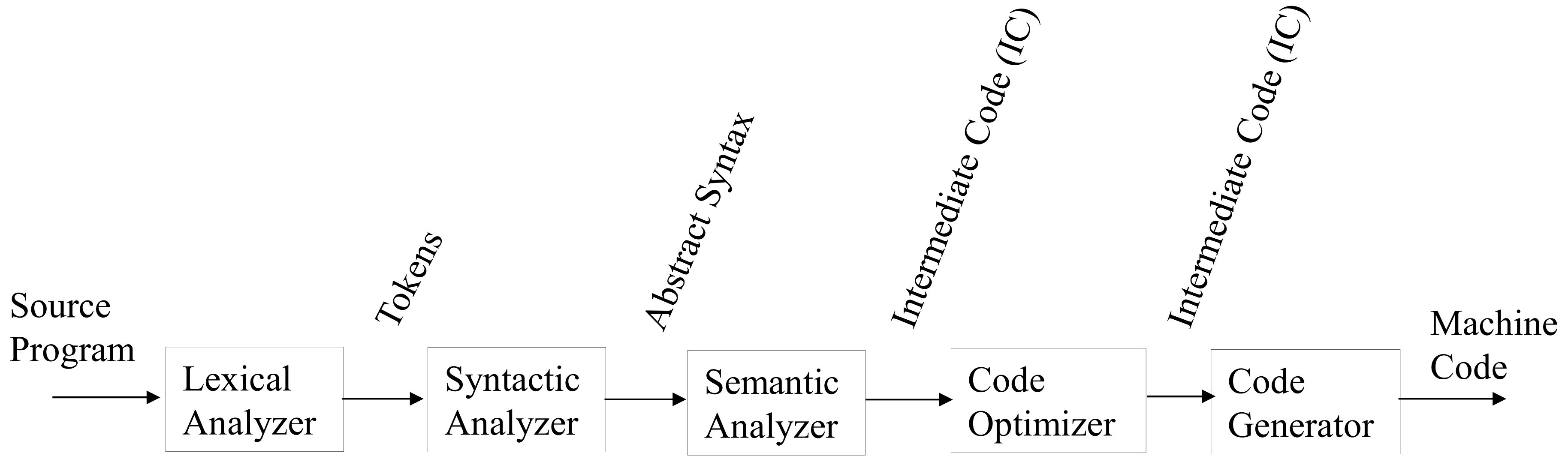
is *not* equivalent to

```
  if (a < x && x < b)
```

But it is error-free!

So, what does it mean?

2.4 Compilers and Interpreters



Lexer

- Input: characters
- Output: tokens
- Separate:
  - *Speed: 75% of time for non-optimizing*
  - *Simpler design*
  - *Character sets*
  - *End of line conventions*

Parser

- Based on BNF/EBNF grammar
- Input: tokens
- Output: abstract syntax tree (parse tree)
- Abstract syntax: parse tree with punctuation, many nonterminals discarded

Semantic Analysis

- Check that all identifiers are declared
- Perform type checking
- Insert implied conversion operators (i.e., make them explicit)

Code Optimization

- Evaluate constant expressions at compile-time
- Reorder code to improve cache performance
- Eliminate common subexpressions
- Eliminate unnecessary code

Code Generation

- Output: machine code
- Instruction selection
- Register management
- Peephole optimization

Interpreter

- Replaces last 2 phases of a compiler
- Input:
  - *Mixed: intermediate code*
  - *Pure: stream of ASCII characters*
- Mixed interpreters
  - *Java, Perl, Python, Haskell, Scheme*
- Pure interpreters:
  - *most Basics, shell commands*

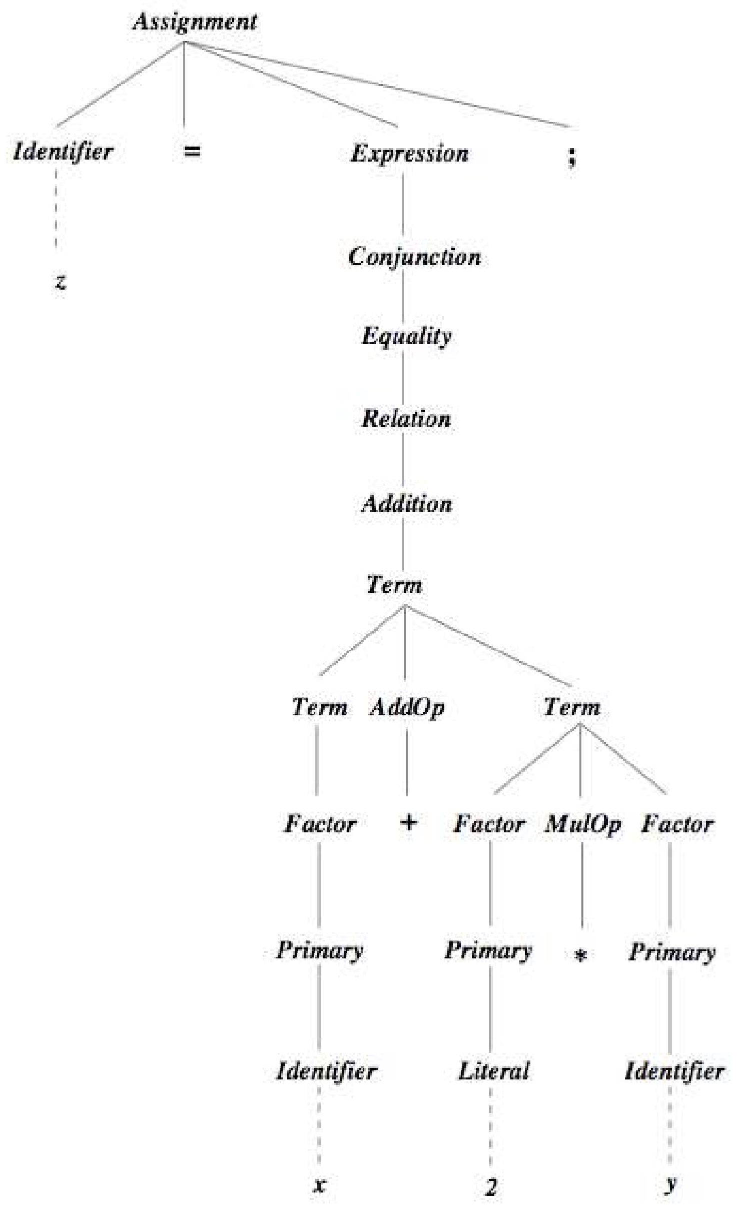
2.5 Linking Syntax and Semantics

- Output: parse tree is inefficient
- Example: Fig. 2.9

Parse Tree for

z = x + 2*y;

Fig. 2.9



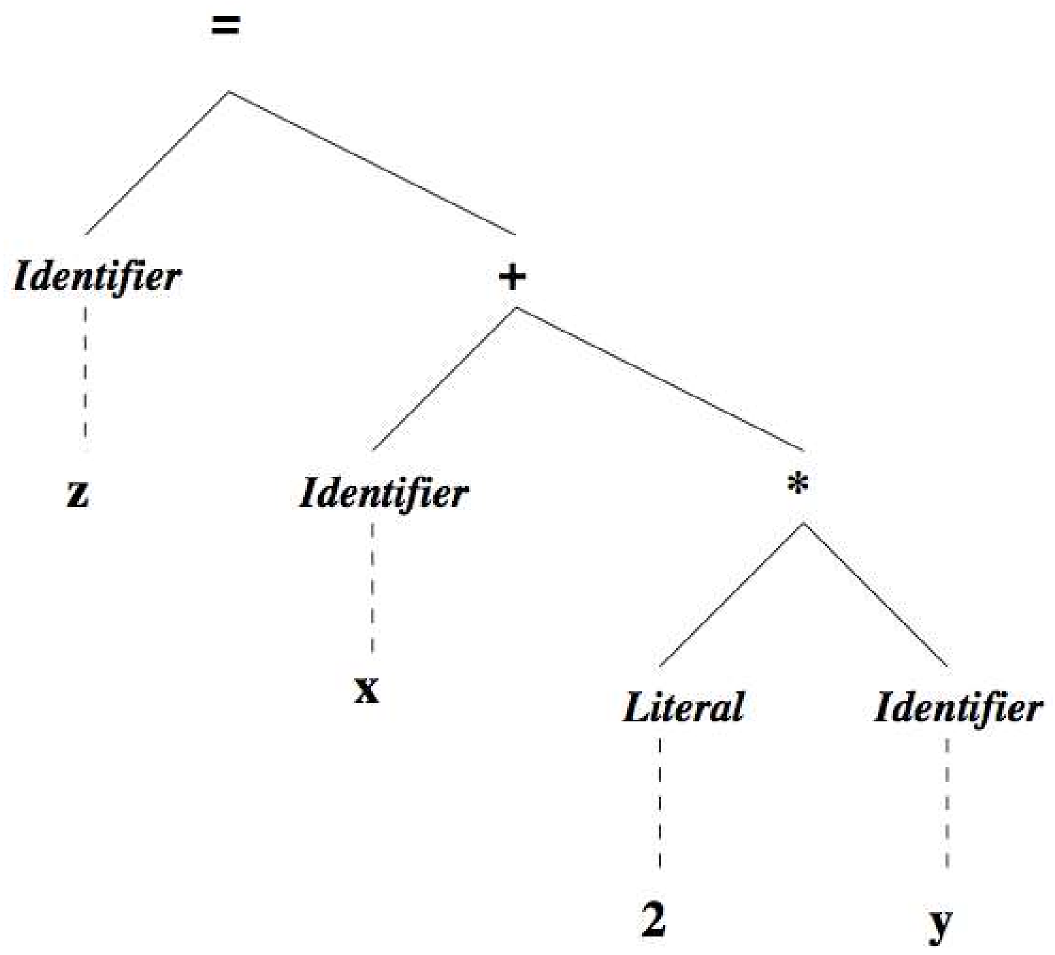
Finding a More Efficient Tree

- The *shape* of the parse tree reveals the meaning of the program.
- So we want a tree that removes its inefficiency and keeps its shape.
  - *Remove separator/punctuation terminal symbols*
  - *Remove all trivial root nonterminals*
  - *Replace remaining nonterminals with leaf terminals*
- Example: Fig. 2.10

Abstract Syntax Tree for

```
z = x + 2*y;
```

Fig. 2.10



Abstract Syntax

Removes "syntactic sugar" and keeps essential elements of a language. E.g., consider the following two equivalent loops:

Pascal

```
while i < n do begin
   i := i + 1;
end;
```

C/C++

```
while (i < n) {
   i = i + 1;
}
```

The only essential information in each of these is
1) that it is a *loop*,
2) that its terminating condition is i < n,
and 3) that its body increments the current value of i.

Abstract Syntax of *Clite* Assignments

Assignment = Variable target*; Expression* source

Expression = VariableRef | Value | Binary | Unary

VariableRef = Variable | ArrayRef

Variable = String id

ArrayRef = String id*; Expression* index

Value = IntValue | BoolValue | FloatValue | CharValue

Binary = Operator op*; Expression* term1, term2

Unary = UnaryOp op*; Expression* term

Operator = ArithmeticOp | RelationalOp | BooleanOp

IntValue = Integer intValue

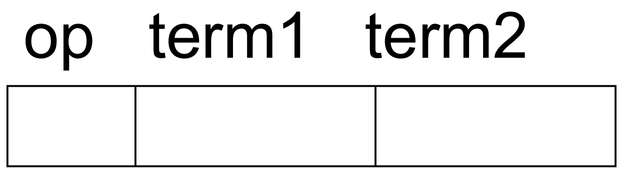
…

Abstract Syntax as Java Classes

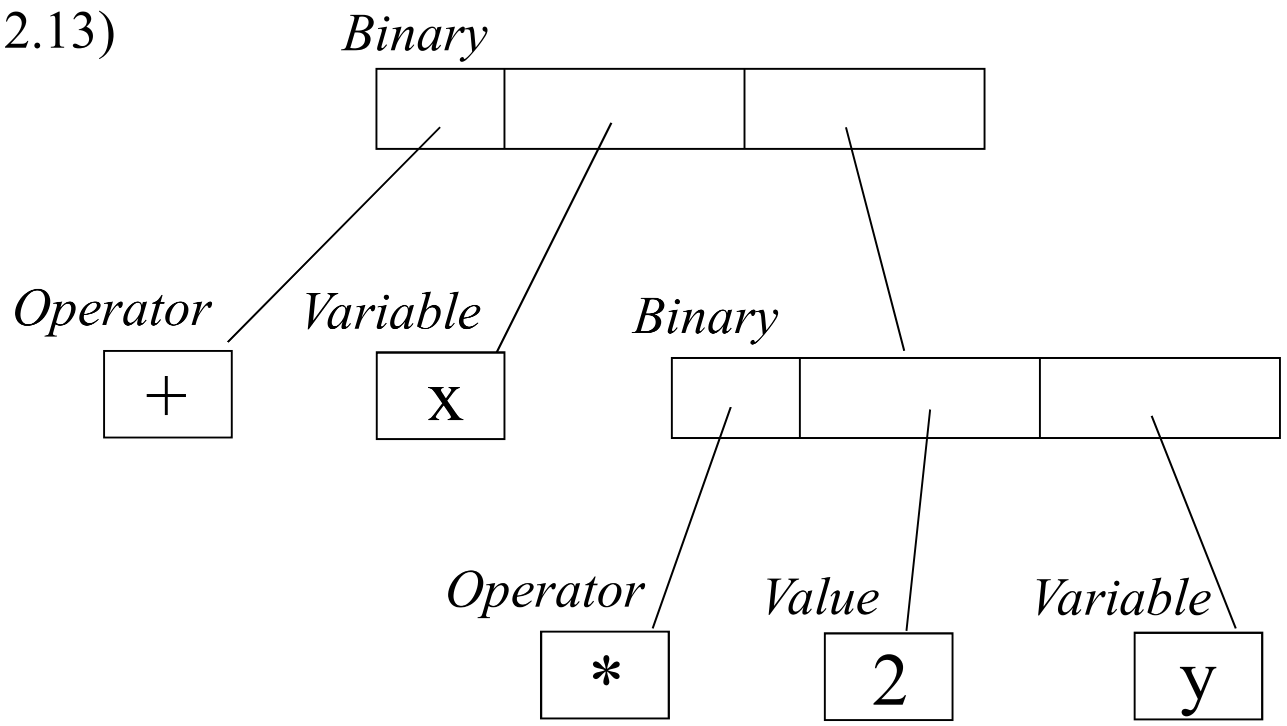
```
abstract class Expression { }

abstract class VariableRef extends Expression { }

class Variable extends VariableRef { String id; }

class Value extends Expression { … }

class Binary extends Expression {

     Operator op;

     Expression term1, term2;

}

class Unary extends Expression {

     UnaryOp op;

     Expression term;

}
```

Example Abstract Syntax Tree

- *Binary* node



- Abstract Syntax Tree
- for x+2*y (Fig 2.13)



Remaining Abstract Syntax of *Clite* (*Declarations* and *Statements*) Fig 2.14

$$
\begin{aligned}
Program &= Declarations\ \texttt{decpart;}\ \ Statements\ \texttt{body;}\\
Declarations &= Declaration^{*}\\
Declaration &= VariableDecl \mid ArrayDecl\\
VariableDecl &= Variable\ \texttt{v;}\ Type\ \texttt{t}\\
ArrayDecl &= Variable\ \texttt{v;}\ Type\ \texttt{t;}\ Integer\ \texttt{size}\\
Type &= \ \texttt{int} \mid \texttt{bool} \mid \texttt{float} \mid \texttt{char}\\
Statements &= Statement^{*}\\
Statement &= Skip \mid Block \mid Assignment \mid Conditional \mid Loop\\
Skip &=\\
Block &= Statements\\
Conditional &= Expression\ \texttt{test;}\ Statement\ \texttt{thenbranch, elsebranch}\\
Loop &= Expression\ \texttt{test;}\ Statement\ \texttt{body}
\end{aligned}
$$